NASA Technical Memorandum 85945

Study of the Mapping of Navier-Stokes Algorithms onto Multiple-Instruction/Multiple-Data-Stream Computers

D. Scott Eberhardt, Donald Baganoff and
K.G. Stevens, Jr.

July 1984

NASA
National Aeronautics and
Space Administration

NASA Technical Memorandum 85945

Study of the Mapping of Navier-Stokes Algorithms onto Multiple-Instruction/Multiple-Data-Stream Computers

D Scott Eberhardt
Donald Baganoff, Stanford University, Stanford, California
K G Stevens, Jr , Ames Research Center, Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

STUDY OF THE MAPPING OF NAVIER-STOKES ALGORITHMS ONTO MULTIPLE-INSTRUCTION/MULTIPLE-DATA-STREAM COMPUTERS

D. Scott Eberhardt,* Donald Baganoff,* and K. G. Stevens, Jr.

Ames Research Center

SUMMARY

Implicit approximate-factored algorithms have certain properties that are suitable for parallel processing. This study demonstrates how a particular computational fluid dynamics (CFD) code, using this algorithm, can be mapped onto a multiple-instruction/multiple-data-stream (MIMD) computer architecture. An explanation of this mapping procedure is presented, as well as some of the difficulties encountered when trying to run the code concurrently. Timing results are given for runs on the Ames Research Center's MIMD test facility which consists of two VAX 11/780's with a common MA780 multi-ported memory. Speedups exceeding 1.9 for characteristic CFD runs were indicated by the timing results.

INTRODUCTION

The purpose of this study was to develop a method for implementing an implicit, approximate-factorization algorithm onto a multiple-instruction/multiple-data-stream (MIMD) computer architecture. This new architecture is representative of the new generation of computers that has just been introduced. To use these machines most efficiently, it is necessary to understand how the particular algorithms map onto the multiprocessor machines. In particular, Ames Research Center is interested in exploring ways to use its recently acquired Cray X-MP (a two-processor MIMD machine) most efficiently. The test results presented here were obtained with an MIMD test facility at Ames consisting of two VAX 11/780's with an MA780 dual-ported memory.

The particular code that was studied on the MIMD test facility was the Pulliam and Steger "AIR3D" code (ref. 1). AIR3D is a three-dimensional, implicit, approximate-factored algorithm which solves the Euler equations or the Navier-Stokes equations with a thin-layer approximation for either laminar or turbulent boundary layers about an axisymmetric, hemispherical nose projectile. Parallel studies have been conducted at Ames to investigate two other widely used computational fluid dynamics (CFD) algorithms: TWING, a two-dimensional potential algorithm, and Rogallo's LES (large eddy simulation) code using spectral methods (refs. 2 and 3). The experience of implementing these three benchmark algorithms onto the MIMD test facility has yielded a clear method for mapping certain CFD algorithms onto MIMD computers.

We first discuss a general approximate-factorization algorithm and some of its properties and then give a detailed discussion of AIR3D, which demonstrates the steps required to transfer a code onto an MIMD machine. Included in the discussion is an outline of the serial code that helps to explain the structure of the concurrent code used. Some of the difficulties encountered in implementing the scheme are presented,

---

*Stanford University

as well as some general thoughts on algorithm-based architectures. Finally, results are presented of the timings obtained by running the concurrent code on the Ames MIMD test facility.

APPROXIMATE FACTORIZATION

The approximate-factorization algorithm is a particular form of the alternating direction, implicit (ADI) algorithm first introduced by Peaceman and Rachford (ref. 4) for two-dimensional problems. The scheme was improved and extended to three dimensions by Douglas (ref. 5), Douglas and Rachford (ref. 6), and Douglas and Gunn (ref. 7). For a hyperbolic set of equations, such as the Euler equations, which we represent by the general form

$$\partial_t q + \partial_x E + \partial_y F + \partial_z G = 0 \tag{1}$$

where q is a vector, $E = E(q)$, $F = F(q)$, and $G = G(q)$, the corresponding finite-difference equations can be expressed in operator notation and written as follows:

$$\mathscr{F}_{xyz} q^{n+1} = \mathscr{L}_{xyz} q^n \tag{2}$$

where the left-hand side is the implicit part and the right-hand side is the explicit part of the algorithm. The operators in equation (2) are general operators that result from the finite-differencing and linearization of the terms containing the E, F, and G differentials. The approximate factorization is introduced as a less computationally costly means of inverting the operator on the left-hand side. The operator is first factored into three separate operators that are spatially independent as follows:

$$\mathscr{F}_{xyz} = \mathscr{F}_x \mathscr{F}_y \mathscr{F}_z + 0(\Delta t^2) \tag{3}$$

and the approximation is introduced by ignoring the second-order terms in equation (3). This allows the introduction of a three-step solution process in which each step inverts an independent spatial operator. Intermediate variables are encountered in this way, but they do not add to the storage requirements since they may overwrite the previous level. The three-step solution is given by

$$\begin{aligned} q^* &= \mathscr{F}_y \mathscr{F}_z q^{n+1} = \mathscr{F}_x^{-1} \mathscr{L}_{xyz} q^n \\ q^{**} &= \mathscr{F}_y^{-1} q^* \\ q^{n+1} &= \mathscr{F}_z^{-1} q^{**} \end{aligned} \tag{4}$$

This spatial decoupling lends itself very nicely to concurrent processing. Since each operator contains derivatives in only one spatial direction, all lines of data in that coordinate can be solved independently. For example, each line of j, where j is the x-index, can be solved independently on every point in the k,ℓ plane, where k and ℓ are the indices of y and z, respectively. Thus, an MIMD machine could, in principle, use as many processors as there are points in each plane, assuming no other restriction.

MIMD IMPLEMENTATION

An MIMD implementation of this spatially decoupled procedure begins as follows. The first step is to compute the right-hand side of equation (2). Because it is explicit and all data at the current time-step are available (in a shareable memory), the data can be divided into several groups. A convenient split is to evenly divide the data along a particular direction, say x, by the number of processors available. If j is the index representing the x-direction and Jmax is the total number of x grid points, then for a two-processor system, one processor can be assigned the points 1 to Jmax/2 and the other Jmax/2 + 1 to Jmax. All processors must be synchronized at the completion of this step before continuing to the implicit integration. This synchronization, following the explicit right-hand-side calculation, is the first of four such synchronizations. The overall procedure is outlined in figure 1.

After all processors have verified completion of their respective segments of the right-hand side, the inversion of $\mathscr{F}_x$ may begin. A single line of j can be inverted at any point in the k,ℓ plane independent of all other lines of j. Thus, the workload can be shared among the several processors into any desired division of the k,ℓ plane. At the completion of the x-sweep a second synchronization must occur before proceeding to the y-sweep.

For the y-sweep, the data base can be split anywhere in the j,ℓ plane to separate the decoupled k-lines for concurrent processing. Upon completion of this sweep, the processors must be synchronized a third time before continuing to the z-sweep. The z-sweep can be split anywhere in the j,k plane, and computation proceeds as before. A fourth and final synchronization is required at the end of the z-sweep to complete a single iteration loop. This loop may have to be repeated 200-600 times before a converged solution is obtained with typical CFD applications.

An example of a simple two-dimensional problem requiring a two-step solution procedure with three synchronizations is shown in figure 2. The figure outlines the process described above for a four-processor system.

SOME OBSERVATIONS

Figure 2 reveals an interesting possibility for hardware implementation of the approximate-factorization algorithm. If a multiprocessor computer system were to be designed specifically for this particular algorithm, then a special memory system could be implemented based on the mesh used in the computation, which follows the "dance hall model." In the case of the two-dimensional problem of figure 2, the computational domain is divided by the four processors and the two sweeps into 16 blocks, as shown in figure 3. However, if one determines which processor accesses which block, one sees immediately that the diagonal blocks are accessed only by a single processor. In fact, the blocks in the computational domain may be associated with matrix elements A_{jk}, where j and k identify the processors that access a block in the two sweeps. Figure 3 exhibits a four-processor implementation using this notation. Thus, for a dedicated approximate-factorization multiprocessor computer system, the memory configuration may be chosen so that a block is accessed by a single processor if $j = k$ or by two processors if $j \neq k$. Since each block is accessed by one processor at a time, the implementation should make use of switches which may be reset by the software on each sweep.

The primary motivation for considering this approach is the problem of memory bus bandwidth. As more processors are added to a common memory bus, timing conflicts grow in number, and processors must wait to access memory. Hardware studies show that four processors on one bus tend to saturate the memory bus, and adding more processors simply degrades processor efficiency rather than improving overall performance. Of course, this varies according to the particular problem being solved. The memory implementation discussed here would circumvent this problem, provided the required switching can be suitably implemented in hardware. Also, in principle, there would be no restriction on how many elements, A_{jk}, are used. Although software may be required to align the data base for each A_{jk} access, which would introduce some additional complexity, this memory implementation would be an interesting possibility for future development.

DESCRIPTION OF CODE

A general discussion of the approximate-factorization algorithm has been presented, and the specific code studied for this report will now be discussed. Our handling of the code AIR3D follows the same general solution procedure described above, but it will be outlined here in more detail. First, we discuss the set of equations and the specific finite-difference operators used. This discussion is not essential to one's understanding of the procedure, provided one is familiar with the general operator notation presented in the previous sections. The specific modifications required for concurrent processing, including a discussion of the required system calls, are given following the code description.

Equation Set

AIR3D is an implicit, finite-difference program for unsteady, three-dimensional flow calculations. It can handle viscous effects and incorporates an algebraic turbulence model as a selected option. The code can also handle arbitrary geometries through the use of a general coordinate transformation. The code is described in detail in a paper by Pulliam and Steger (ref. 1), which will only be summarized here.

The three-dimensional, nonsteady Navier-Stokes equations can be tranformed and written for an arbitrary curvilinear space, while retaining the strong conservation-law form, without undue increased complexity of the governing set. The following form shows the resulting equations when transformed from x,y,z,t space to ξ,η,ζ,τ space:

$$\partial_\tau q + \partial_\xi (E + E_v) + \partial_\eta (F + F_v) + \partial_\zeta (G + G_v) = 0 \tag{5}$$

where

$$q = J^{-1}\begin{vmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{vmatrix}, \quad E + J^{-1}\begin{vmatrix} \rho U \\ \rho uU + \xi_x p \\ \rho vU + \xi_y p \\ \rho wU + \xi_z p \\ (e + p)U - \xi_t p \end{vmatrix}, \quad F = J^{-1}\begin{vmatrix} \rho V \\ \rho uV + \eta_x p \\ \rho vV + \eta_y p \\ \rho wV + \eta_z p \\ (e + p)V - \eta_t p \end{vmatrix}$$

$$G = J^{-1}\begin{vmatrix} \rho W \\ \rho uW + \zeta_x p \\ \rho vW + \zeta_y p \\ \rho wW + \zeta_z p \\ (e + p)W - \zeta_t p \end{vmatrix}$$

and

$$U = \xi_t + \xi_x u + \xi_y v + \xi_z w$$

$$V = \eta_t + \eta_x u + \eta_y v + \eta_z w$$

$$W = \zeta_t + \zeta_x u + \zeta_y v + \zeta_z w$$

The quantities U, V, and W are the contravariant velocities written without metric normalization. Note that this general transformation includes the possibility of a moving grid. The viscous terms will not be presented here but can be found in many papers on the subject (refs. 1, 8, 9). In this formulation, the Cartesian velocity components u,v,w are nondimensionalized with respect to the free-stream speed of sound a_∞, the density ρ is normalized with respect to ρ_∞, and the total energy e is normalized with respect to $\rho_\infty a_\infty$. Pressure is given in terms of these variables by

$$p = (\gamma - 1)[e - 0.5(u^2 + v^2 + w^2)] \tag{6}$$

The metric terms themselves are defined in detail in reference 1.

This program makes use of a thin-layer approximation throughout, resulting in fewer grid points and less computation. The thin-layer approximation uses boundary-layer-like coordinates and ignores viscous terms associated with small velocity gradients. Therefore, if the ξ and η coordinates are chosen to lie parallel to the body surface, only the ζ viscous terms will be included. This is identical to a boundary-layer model in which streamwise viscous terms are ignored. Thus, this approximation requires grid refinement in only the ζ, or perpendicular, direction. The new set of equations simplifies to

$$\partial_\tau q + \partial_\xi E + \partial_\eta F + \partial_\zeta G = Re^{-1}\partial_\zeta S \tag{7}$$

where

$$S = J^{-1}\begin{vmatrix} 0 \\ \mu(\zeta_x^2 + \zeta_y^2 + \zeta_z^2)u_\zeta + (\mu/3)(\zeta_x u_\zeta + \zeta_y v_\zeta + \zeta_z w_\zeta)\zeta_x \\ \mu(\zeta_x^2 + \zeta_y^2 + \zeta_z^2)v_\zeta + (\mu/3)(\zeta_x u_\zeta + \zeta_y v_\zeta + \zeta_z w_\zeta)\zeta_y \\ \mu(\zeta_x^2 + \zeta_y^2 + \zeta_z^2)w_\zeta + (\mu/3)(\zeta_x u_\zeta + \zeta_y v_\zeta + \zeta_z w_\zeta)\zeta_z \\ (\zeta_x^2 + \zeta_y^2 + \zeta_z^2)[0.5\mu(u^2 + v^2 + w^2)_\zeta + \kappa Pr^{-1}(\gamma - 1)(a^2)_\zeta] \\ \quad + (\mu/3)(\zeta_x u + \zeta_y v + \zeta_z w)(\zeta_x u_\zeta + \zeta_y v_\zeta + \zeta_z w_\zeta) \end{vmatrix}$$

An algebraic turbulence model is also incorporated in AIR3D which makes use of the method of Baldwin and Lomax (ref. 10).

Algorithm

The finite-difference scheme used in AIR3D is the implicit approximate-factorization algorithm of Beam and Warming (ref. 8). The scheme was chosen to be implicit to avoid the restrictive stability bounds of explicit methods when applied to small grid spacings. The delta form of the algorithm, where incremental changes in quantities are calculated, is used to reduce computational errors and, in addition, is a convenient choice for steady-state solutions.

Central differencing is used for all three directions. The finite-difference equations are spatially split so that three separate one-dimensional problems are solved at each time-step. The central differencing yields block-tridiagonal matrices which are inverted in each spatial coordinate. This decoupling of the spatial coordinates provides the principal motivation for considering parallel processing as a means of carrying out the calculations in an efficient way.

The approximate factorization of the finite-difference algorithm results in the following set of finite-difference equations:

$$(I + h\delta_\xi \tilde{A}^n - \varepsilon_i J^{-1}\nabla_\xi\Delta_\xi J)(I + h\delta_\eta \tilde{B}^n - \varepsilon_i J^{-1}\nabla_\eta\Delta_\eta J)(I + h\delta_\zeta \tilde{C}^n - hRe^{-1}\delta_\zeta \tilde{M}^n - \varepsilon_i J^{-1}\nabla_\zeta\Delta_\zeta J)\Delta q^n$$

$$= -\Delta t(\delta_\xi E^n + \delta_\eta F^n + \delta_\zeta G^n - Re^{-1}\delta_\zeta S^n) - \varepsilon_e J^{-1}[(\nabla_\xi\Delta_\xi)^2 + (\nabla_\eta\Delta_\eta)^2 + (\nabla_\zeta\Delta_\zeta)^2]Jq^n \tag{8}$$

where $\Delta q^n = q^{n+1} - q^n$ and $h = \Delta t$ for first-order Euler time-differencing, or $h = \Delta t/2$ for second-order trapezoidal time-differencing. The finite-difference operators ∇, Δ, and δ are explained in the original paper (ref. 1); they are widely used. The matrices $\tilde{A}^n$, $\tilde{B}^n$, and $\tilde{C}^n$ are time-linearizations of E^{n+1}, F^{n+1}, and G^{n+1}, respectively. The coefficient matrix $\tilde{M}^n$ is obtained by a Taylor-series expansion of the viscous vector S^{n+1}. These matrices will not be presented here but can be found in reference 1. Numerical damping terms are added to improve the stability, and are selected to be second-order accurate to maintain the block-tridiagonal nature of the implicit part of the code; they are fourth-order accurate in the explicit right-hand side of the code.

In terms of the operator notation introduced in equations (4), each operator becomes a block-tridiagonal matrix in this formulation. The operators are each defined by

$$\mathscr{F}_x = (I + h\delta_\xi \tilde{A}^n - \varepsilon_i J^{-1}\nabla_\xi\Delta_\xi J)$$

$$\mathscr{F}_y = (I + h\delta_\eta \tilde{B}^n - \varepsilon_i J^{-1}\nabla_\eta\Delta_\eta J \tag{9}$$

$$\mathscr{F}_z = (I + h\delta_\zeta \tilde{C}^n - \varepsilon_i J^{-1}\nabla_\zeta\Delta_\zeta J - hRe^{-1}\delta_\zeta M^n)$$

The block-tridiagonal operators $\mathscr{F}_x$, $\mathscr{F}_y$, and $\mathscr{F}_z$ are spatially decoupled and so can be inverted independently. Each operator contains derivatives in only one direction; for example, each $\mathscr{F}_x^{k\ell}$ is independent of every other $\mathscr{F}_x^{k\ell}$; $\mathscr{F}_x^{k\ell}$ is thus

inverted at each η and ζ coordinate independently. In a Kmax by Lmax MIMD processor array, a single sweep of each η,ζ processor would invert the entire $\mathscr{F}_x$ operator. The Ames MIMD machine uses only two processors so the workload was shared equally; consequently, each processor inverted Kmax/2 by Lmax block tridiagonals. The explicit operator, $\mathscr{L}_{xyz}$, can be handled in any convenient manner since it is completely explicit, and all the required data are available at each time-step. The division used in this study was a simple Jmax/2 split so that the data base was divided into two blocks of Jmax/2 by Kmax by Lmax points.

It should be noted that this decoupling is a feature of the approximate factorization and is not associated with the central differencing used in AIR3D. Thus, any algorithm exhibiting spatial decoupling should allow the use of the same sort of parallelism.

Running the Serial Code

The conventional method of running the code AIR3D on a serial machine will now be described. This will help to give a better understanding of the modifications introduced to allow the use of concurrency. Figure 4 shows a flowchart of the serial code, and figure 5 shows a detailed breakdown of tasks in AIR3D with subroutine names from the program.

The initial segment of the code (see fig. 4) includes the input routines, initialization routines, and the grid-generation routines. The input routines set the angle of attack, Mach number, and other important flow variables. Switches are also set which specify the grid option and initialization option used. Also, the switches determine whether viscous effects are included and whether they are laminar or turbulent. The initialization routines allow the choice of an impulsively started solution or a start-up from a previous solution which is obtained from a file. The grid-generation routine allows the selection of either a grid stored in a file or the default grid (a hemispherical nose with a cylindrical afterbody), which it calculates. After this initial segment is run, the program enters the main iteration loop.

The main iteration loop contains the code section which updates the solution by one iteration step. This segment begins by calculating the boundary conditions explicitly. Then the right-hand-side operator is calculated, followed by the explicit smoothing. At this point, the residual operator is available (at steady state the right-hand side becomes zero) and so convergence is tested by calculating the L_2 norm. Optional output routines give diagnostic information, such as a pressure distribution, when requested. The final step in the main iteration loop is the implicit integration. This requires three sweeps through the data base. In the ξ-sweep a block tridiagonal is inverted for each point in the η,ζ plane. Likewise, a block tridiagonal is inverted for each point in the ξ,ζ and the ξ,η planes for the η-sweep and ζ-sweep, respectively. This main iteration loop accounts for most of the CPU time, since it is repeated 200-600 times for typical solutions and it is computationally intensive.

The final portion of the code contains the output routines. This portion places the output data into output files for future data processing.

The original code was written to run on the CDC 7600 since the code is too computationally intensive to run on the VAX 11/780. Because the goal of the present study was to develop initial experience with an MIMD test bed consisting of two VAX 11/780's with an MA780 memory, the Fortran code was preprocessed to eliminate CDC 7600 Fortran

extensions, and the grid density was reduced to make the code compatible with the available memory. This modification will be described below. Also, owing to the limited available computational time, no runs were carried through to a converged solution, which would have required the exclusive use of a machine for several days. Nevertheless, because a converged solution was not required and only timing data were needed, timing predictions could be obtained quite accurately for large times from sets of runs that could be carried out in reasonable time periods.

Running the Concurrent Code

The concurrent code, called MAIN, runs with several spawned processes operating on each machine. The program breakdown is shown in figures 6 and 7; subroutine names are given in figure 7. The processes on each machine must communicate with each other and with the processes on the other machines through global sections in shared memory. The specific mechanisms for this interprocess communication are calls to the VAX VMS system-service routines and run-time library functions. They will be presented by their function names; they are described in detail in the VAX system service manual (ref. 11). Since these service calls are machine-dependent, the source code is not transferable to other MIMD machines without suitable translation of the system service calls. A diagram showing the process relationships to each other and memory is shown in figure 8. The individual sections of this code will now be described.

The first section of MAIN not only contains all of the elements of the initial part of the serial code, but also includes some important elements of the parallel processing. In addition to initialization, etc., the first section sets up the global sections and flag clusters in shared memory and creates subprocesses, which represent the concurrent code on that machine. The second processor, referred to here as the slave processor, has a similar bookkeeping program, called SLAVE, for identical parallel processing functions; however, it contains none of the routines of the original serial code. Calls to MAPPRM, a function call containing the system service SYS$MGBLSC, reserve global sections that are required for global common blocks. These blocks, which reside in shared memory, are given the logical names SHRMEMO:name. Table 1 identifies by name and function the common blocks used in each process and their level of protection (local or global). The system service call SYS$ASCEF sets up flag clusters for interprocess communication. Two flag clusters have been allocated for this program: one represents synchronization communication between subprocesses and the main processes, and the other represents semaphore, or flag, communication between the main processes on the two processors. Finally, a call to LIB$SPAWN, a run-time library routine, creates the subprocesses that represent the concurrent portions of the code. These subprocesses are given the names SUBPRAIR3D$01-04 and run the program STEPUP and SUBRHS. The numerical extension in the subprocess name, 01-04, is the identification number used in the program to define which portions of data to work with.

All programs that need data from the shared memory sections must map global sections to the shared memory before they can access it. All programs must also acknowledge, or associate with, the flag clusters which are an essential part of the interprocess communication. Once the initial setup for parallel processing has been completed and MAIN has completed the initialization routines described in the serial code, the grid metric terms must be passed to the other processor's local memory. This is due to a hardware restriction and will be discussed in detail below. MAIN notifies the slave processor when it has reached this point by setting a flag (SYS$SETEF) in the semaphore flag custer. Slave, which is idle during the parallel processing setup (waiting for a flag, SYS$WAITFR), completes the transfer, clears the

flag (SYS$CLREF) for future use, and sets another flag to notify MAIN that the transfer is complete. Initialization is now complete for MAIN and SLAVE. Most of this first section of the code is basically sequential, and the remainder does not warrant the effort to extract parallel segments. The slave processor therefore remains mostly idle during this setup phase.

Two processes are spawned on each processor during the setup phase. The programs submitted are called STEPUP and SUBRHS, which contain the solvers for the implicit integration and for the right-hand side, respectively. After these two jobs are created at the beginning of MAIN and SLAVE, they run through an initialization phase which maps the global sections, associates with flag clusters, and identifies which half of the data it will work on. This identification is obtained by a call to SYS$GETJPI which returns the subprocess name used by the system (SUBPRARI3D$xx). The numerical extension is extracted which becomes its ID number. The initial setup is very short and when complete, the processes hibernate (a call to SYS$HIBER) until awakened by the main programs.

The main iteration loop constitutes most of the parallelism found in the code. Initially, boundary data are calculated serially, because very little improvement in overall speedup would be gained and it would not offset the effort needed to parallelize it. This could be implemented along with the right-hand side at a future time, but it would require rather extensive rewriting of that portion of the code. When the right-hand-side operator is ready to be calculated, MAIN notifies SLAVE, through a semaphore flag, and each wakes the subprocess SUBRHS on its respective machines (by a call to SYS$WAKE). At this time the main programs, MAIN and SLAVE, go into a wait state until a flag code is set by the subprocesses (SYS$WFLAND). The two processes, called SUBPRAIR3D$03 ($04), now run the concurrent program SUBRHS through a cycle which calculates the right-hand-side operator, adds the explicit smoothing, and sets an exit flag, before returning to a ready state at the beginning of the code and hibernating there. When both the subprocesses have set their event flags, the main programs are reactivated. First, the main processes clear the event flags, and then MAIN continues with the calculation of the residual and calls optional output routines, as in the serial code. The slave processor remains idle for this short period.

The implicit integration is a much more complex process, because three cycles are needed to complete a single iteration. Again, as for the right-hand-side calculation, MAIN notifies SLAVE to begin. Both main processes wake up the subprocesses, now called SUBPRAIR3D$01 (02), which run the program STEPUP concurrently. During the first cycle, the direction flag (IDIR) is set to 1, signifying a ξ-sweep. After each subprocess has completed its half of the calculation, synchronization is introduced by setting event flags to notify the main processes, as with SUBRHS. The cycle then repeats with the direction flag set to 2 and then 3, signifying η- and ζ-sweeps. A single iteration loop has been completed at this point, requiring four synchronizations. The synchronizations between the directional sweeps in STEPUP are required so that the intermediate starred variables in the three-step-solution process of equation (4) will be at the proper stage when the processors call upon them. The program STEPUP loops back to the beginning of the code, to the SYS$HIBER call, where it waits until called upon again. This point is also the starting point for each of the ξ-, η-, and ζ-sweeps.

The final section of the code is identical to the serial code except that MAIN must now notify SLAVE to exit and delete the subprocesses. MAIN also deallocates the shared memory global sections and flag clusters. For most of the output routines, the slave processor remains idle.

A summary of the system calls that were used in this implementation is presented to assist in making the code transportable to other MIMD machines. First, the program must be allowed to control mapping of data into shared memory, so that each processor can access global common blocks, yet protect local data. Next, a mechanism for setting up flag clusters must exist with the following communication devices: (1) a wait for a particular flag device; (2) a wait for a logical AND of the flag cluster with a variable mask; (3) a clear flag; and (4) a set flag command. Services that allow process hibernation and waking are also desirable, although flag communications could replace these calls. Lastly, a routine for spawning subprocesses would be required to run this particular implementation. The subprocess creation is not necessarily required since the code could reside in the MAIN and SLAVE codes; however, this would not separate tasks conveniently. If the subprocess implementation is used, the numerical ID extension to the subprocess name is convenient for identification. Regardless of implementation, some mechanism for process identification must exist (processor number, etc.), so that the programs will know which data they are assigned. Once these system calls are available, AIR3D can be transferred to any MIMD machine with the proper translation of the VAX system service calls in the current program.

DIFFICULTIES OF IMPLEMENTATION

Several trade-offs and compromises were made in this concurrent implementation of AIR3D. Most of these were a result of the limitations imposed by the particular MIMD test facility employed. First, because the code is computationally intensive, converged solutions could not be generated on the two VAX's. The time required to reach a properly converged solution would have required the total dedication of both the MERCURY and JUPITER computers at Ames for several days. Since this amount of run time was not even considered, the test cases were run for only 10 to 20 iterations to get sample timings.

Another rather severe restriction encountered was the limited size of the MA780 dual-ported memory in the system. Only 1/4 Mbyte was available, and all shared memory requirements had to fit within this memory size. Because the code is three-dimensional and each grid point has 14 variables associated with it, the shared memory was quickly used up and two steps had to be taken to tailor the problem to the limited memory. First, the grid metrics were removed from shared memory and a copy was placed in each processor's local memory. This eliminated 3 of the 14 variables required. It must be noted that in an unsteady problem, where the grid metrics change dynamically, this procedure would not be allowed. The passing of the grid metrics from the main processor to the slave processor occurs during the initialization routines as described above. The second step taken was to reduce the grid density. The normal default case for the hemispherical nose, cylindrical afterbody geometry was an array of $48 \times 12 \times 20$ points for the inviscid case and a $30 \times 18 \times 30$ array for the viscous case. All cases in this study used a $20 \times 10 \times 20$ array which results in 40% fewer grid points. At this level of coarseness, the code became unstable after a large number of iterations and no attempt was made to seek a converged solution. This was not a serious limitation because the objective of this study was to obtain a run-time comparison between a serial and an MIMD configuration, and for this comparison convergence is not a necessary condition. These two steps would not have been required on a machine with a larger shared memory, such as the Cray X-MP.

Another approach was initially taken but then dropped because it was clear that a converged solution was not required for this study. The approach considered was to place only the data that were required by the slave processor in the shared memory,

which meant that only half of the data had to be in shared memory at one time. However, this also meant the data had to be reformatted and shifted at least twice for each iteration. This reformatting added an unfair burden to the timings of the concurrent code. Since this reformatting was not a consequence of the MIMD architecture but a result of the limited memory available, reformatting represented an undesirable approach. One clear advantage that this approach brought to light was that the slave processor required only a generic solver for all three sweeps of the implicit integration. This resulted in a compact code which, on certain vector machines, could speed up the computation time of the slave processor considerably.

In the original formulation of the program, metric derivatives were calculated as needed to avoid the extra memory space required to store them. The code was written to calculate all of the metric derivatives along a particular line when the subroutine was called. This presented no problem with the implicit part of the code since all lines of data were decoupled. However, with the explicit part of the code, when the metric derivatives crossing the division between the two data halves were calculated, some overlapping data were needed in the calculation, and extra work was necessary given the current code structure. The amount of overlap required was two points, for the fourth-order finite-differencing used in this code, since each half of the explicit calculation must "see" into the other half of the data a distance of two points.

RESULTS

Results in the form of timing measurements will now be presented and discussed. These timings were performed on the Ames MIMD test facility run as a single-user system. This allowed for the use of all the memory available, with only the operating system competing for CPU time. The two options tested were the Euler equations and Navier-Stokes equations with the turbulent, thin-layer approximation.

Three timing measurements are presented for the two flow solvers. The three timings include progressively more hardware/operating system penalties. The measurements were made to demonstrate the variations in timing speedups that are found for different computer environments. The three measurements are for CPU task timing, total CPU timings, and real-time (stopwatch) timings. Speedup as used here is defined by

$$\text{Speedup} = \frac{t_{\text{serial}}}{t_{\text{concurrent}}} \tag{10}$$

The first set of timings, the CPU task timings, was made by recording the CPU time for each element, or task, of the program. The tasks are defined in a manner consistent with the previous discussion of the code. They include the setup, which is serial, both the serial and concurrent parts of the explicit calculation and the implicit integration, the boundary conditions and residual calculation, and, finally, the output routines. The serial portions of the explicit calculation and the implicit integration are primarily overhead, required for parallel processing. This timing procedure makes it easy to separate the serial and concurrent timings for extrapolating speedup for a larger number of iterations. Tables 2 and 3 compare the serial timings and the concurrent timings for the Euler and Navier-Stokes solvers, respectively. The data presented are representative of all the iterations, since the task timings for each iteration were found to be very close, although not identical. From

these timings it is clear that within the main iteration loop, a significant improvement in computation time is achieved. The tasks that calculate the right-hand-side operator and invert the left-hand-side operator show speedups of nearly 2.0, with very little overhead. The main iteration loop showed a speedup of 1.905 for the Euler solution and 1.914 for the Navier-Stokes solution. These results demonstrate that the overall speedup attained for this particular implementation is quite good for a single-iteration cycle. The single-iteration cycle represents a respectable asymptotic limit for typical numbers of iterations required in CFD applications that include initialization and output routines. Plots of speedup versus number of iterations are shown in figures 9 and 10. These curves were computed using the following formula:

$$\text{Speedup} = \frac{[t_{setup} + n(t_{BC} + t_{RHS} + t_{LHS}) + t_{output}]_{serial}}{[t_{setup} + n(t_{BC} + t_{RHS} + t_{LHS}) + t_{output}]_{concurrent}} \qquad (11)$$

An interesting observation was made in developing these timings: the two processors used yielded different timings for the same concurrent task; the timings were found to vary by as much as 5%. However, each processor was consistent with its own timings. As a result, it was decided to use the task timings for the extrapolated MIMD performance curves from the same processor that executed the serial code. Another observation made with these timings was that some tasks are not penalized properly for certain overhead. For example, no task is charged for the CPU time required to wake a process or cause it to hibernate. In view of this, care was taken so that most parallel-processing overhead was properly charged.

The next two sets of timings to be presented represent the total time spent in executing the code (including all overhead), but exclude penalties for work done by the operating system in job management, etc. These results would be representative of nontime-shared machines, where no job management interruptions are allowed, and where all processors operate at the same speed. They also provide a check on the previous timings. Tables 4 and 5 show results for the Euler and Navier-Stokes solutions, respectively. This time is simply the sum of the CPU time for the main process and for each of the subprocesses. Again, for this implementation, where each processor has a slightly different speed, the speedup is measured by using the data for the same processor that was used for the serial code. These results are slightly lower than the previous results, as seen in figures 9 and 10, since all program-related overhead was properly charged.

The last set of timings are "stopwatch-style" timings; they are presented in tables 6 and 7. The measured time represents the total elapsed time from initial start-up of the job to its conclusion. This is the actual speedup in turnaround time that one could expect for this system. This number includes all job accounting overhead from the operating system and the difference in speed of the two processors. This timing, however, is very machine-dependent and, given the MIMD test bed used, it can be assumed to represent the low end of the speedup (see figs. 9 and 10). The difference between this timing and the timings of the first method represents the improvement that can be made within a given computer environment.

All three timings show that the Navier-Stokes solution, with the turbulent, thin-layer approximation, yielded a greater speedup than the Euler solution. This is a consequence of the greater number of calculations needed for the Navier-Stokes solution that appear in the parallel portions of the code. An even greater speedup of both solvers could be achieved if the additional effort were taken to parallelize the boundary conditions and residual calculations. These two portions of code are located

in the main iteration loop and are, therefore, a significant cause of inefficiency. The initial setup and final output routines represent such a small fraction of the total CPU time for the case of a larger number of iterations, which is the more realistic case, that it is not useful to devote any effort to extracting any parallelism they may contain. The timings reported here represent speedups that are obtainable using standard programming practice.

CONCLUSIONS

A significant amount of parallel code has been identified in a standard benchmark CFD code. The code uses an approximate-factored algorithm that, in a very straightforward manner, can be run on a concurrent processing computer. This implicit algorithm was shown to achieve a speedup of greater than 1.9 on a two-processor system for representative solutions and to do so without an undue amount of effort. The general approximate-factored algorithm appears to be a very good candidate to run on the new generation of MIMD computers.

The computer environment encountered in this study brought to light some features that should be considered. In this study it was found that the two processors used required different execution times. Although this would not be significant on a machine like the Cray C-MP, it made significant differences on the Ames MIMD test facility. The conclusion drawn from this observation is that concurrent algorithms should strive to be asynchronous. This would eliminate overhead from the different processor speeds. The approximate-factored algorithm is not an algorithm that can be run asynchronously, so a more complex scheme of balancing processors would be necessary.

Some stumbling blocks at the beginning of the research helped to show the importance of memory management. Local and shared memories must be separate and protected. This requirement led to the significant interaction between the programmer and the operating system/computer architecture which ideally should be eliminated in an operational computer. Solutions to this problem include designing algorithms that either use only shared memory or use a minimal amount of shared memory, where asynchronous passing of influence parameters is used instead of code variables themselves. Another approach would be to design additional language constructs and make them available to the programmer who helps control memory protection. A sophisticated operating system and compiler that could handle these high-level language constructs would be required. No matter which approach is taken, it must eliminate the need for the programmer to access the operating system for memory management.

A whole new area of research can be formed in concurrent algorithm development. Most work in the past has been of an explicit nature, but this study has shown that even implicit algorithms possess a significant amount of parallelism. Research is being initiated on asynchronous algorithms meeting most of the requirements developed above.

REFERENCES

1. Pulliam, T. H.; and Steger, J. L.: On Implicit Finite-Difference Simulations of Three Dimensional Flow. AIAA Paper 78-10, 1978.

2. Holst, T. L.; and Thomas, S. D.: Numerical Solution of Transonic Wing Flow-fields. AIAA J., vol. 21, no. 6, June 1983.

3. Rogallo, R. S.: Numerical Experiments in Homogeneous Turbulence. NASA TM-81315, 1981.

4. Peaceman, D.; and Rachford, H.: The Numerical Solution of Parabolic and Eliptic Differential Equations. SIAM J., vol. 3, 1955.

5. Douglas, J.: On the Numerical Integration of $U_t + U_{xx} + U_{yy}$ by Implicit Methods. SIAM J., vol. 3, 1955.

6. Douglas, J.; and Rachford, H.: On the Numerical Solution of the Heat Conduction Problems in Two and Three Space Variables. Trans. Am. Math. Soc., vol. 82, 1956, pp. 421-439.

7. Douglas, J.; and Gunn, J.: A General Formulation of Alternating Direction Methods. Numer. Math., vol. 6, no. 5, 1964.

8. Beam, R. M.; and Warming, R. F.: An Implicit Factored Scheme for the Compressible Navier-Stokes Equations. AIAA J., vol. 16, no. 4, Apr. 1978.

9. Steger, J. L.: Implicit Finite-Difference Simulation of Flow About Arbitrary Two-Dimensional Geometries. AIAA J., vol. 16, no. 7, July 1978.

10. Baldwin, B. S.; and Lomax, H.: Thin Layer Approximation and Algebraic Model for Separated Turbulent Flows. AIAA Paper 78-257, 1978.

11. VAX/VMS System Services Reference Manual (version 3.0). Digital Equipment Corporation, Maynard, Mass., 1982.

TABLE 1.- COMMON BLOCK PROTECTION

Common block	Function	MAIN	SLAVE	STEPUP	SUBRHS
BASE	Constants, switches	G	G	G	G
GEO	Grid constants	L			
READ	I/O switches	L			
VIS	Viscous constants	G		G	G
VARS	Solution vector	G		G	G
VARO	Previous solution	G	G	G	G
VAR1	Grid metrics	G*	G*	G*	G*
VAR3	Metric derivatives	L		L	L
COUNT	Iteration count	L			
PPRCSS	Parallel processing	G	G	G	G
PLOT	Plot switches	L			
TURMU	Turbulence stresses	G		G	G
MUKIN	Temperature	G		G	G
FS	Free stream	G		G	G
BTRI	Matrix coefficient			L	
RHS	RHS switches				L

Notes: G = global (shared memory); L = local; G* - global on each processor.

TABLE 2.- TASK TIMINGS: EULER EQUATIONS

Task	Time-MIMD code, sec		Time-serial code, sec	Speedup
	Serial	Concurrent		
Setup	6.58		6.03	0.916
RHS	.02	4.02	7.78	1.926
Residual + BC	.74		.74	1.0
LHS	.09	13.64	26.76	1.949
Output (optional)	(1.64)		Not measured	Assume 1.0
Time/iteration	(.85)	(17.66)	(35.26)	1.905
Output routines	3.24		3.29	1.015

$$\text{Speedup} = \frac{t_{setup} + n(t_{BC} + t_{RHS} + t_{LHS}) + t_{output}}{t'_{setup} + n(t'_{BC} + t'_{RHS} + t'_{LHS}) + t'_{output}}$$

n - iterations
t - serial
t' - MIMD

Examples:

Iterations	Speedup	Iterations	Speedup
1	1.574	15	1.872
2	1.705	50	1.895
5	1.813	100	1.900
10	1.857	400	1.904

TABLE 3.- TASK TIMINGS: NAVIER-STOKES EQUATIONS

Task	Time-MIMD code, sec		Time-serial code, sec	Speedup
	Serial	Concurrent		
Setup	6.57		6.07	0.924
RHS	.02	7.84	15.15	1.927
Residual + BC	.84		.84	1.0
LHS	.08	15.47	30.50	1.972
Output (optional)	(1.64)		Not measured	Assume 1.0
Time/iteration	(.94)	(23.31)	(46.42)	1.914
Output routines	3.24		3.22	.994

Examples:

Iterations	Speedup	Iterations	Speedup
1	1.635	50	1.906
2	1.751	100	1.910
5	1.842	400	1.913
10	1.876		
15	1.889		
25	1.899		

TABLE 4.- TOTAL CPU TIMINGS FOR EULER EQUATION

Number of iterations	t_{MAIN}, sec	t_{RHS}, sec	t_{LHS}, sec	t_{MIMD}, sec	t_{serial}, sec	Speedup
1	11.60	4.21	13.67	29.48	45.32	1.537
1	11.79	4.23	13.74	29.76		1.523
2	14.25	8.41	27.36	50.02	82.70	1.653
5	16.10	20.98	67.95	105.03	187.36	1.784
10	22.60	42.04	132.38	197.02	365.60	1.856
15	28.03	63.03	204.58	295.64	544.49	1.842
25	33.45	104.79	338.04	476.28	897.78	1.885

TABLE 5.- TOTAL CPU TIMINGS FOR NAVIER-STOKES EQUATION

Number of iterations	t_{MAIN}, sec	t_{RHS}, sec	t_{LHS}, sec	t_{MIMD}, sec	t_{serial}, sec	Speedup
1	11.62	7.84	15.47	34.93	56.70	1.623
2	13.93	15.69	30.87	60.49	105.07	1.737
5	16.19	39.15	77.15	132.49	244.24	1.843
10	20.58	78.17	155.13	253.88	478.13	1.883
25	33.36	196.72	386.95	617.03	1173.25	1.901

TABLE 6.- STOPWATCH TIMINGS FOR EULER EQUATION

Number of iterations	t_{MIMD}, sec	t_{serial}, sec	Speedup
1	44.61	46.14	1.034
1	36.98		1.248
2	58.10	82.70	1.423
5	114.68	188.32	1.642
10	213.66	366.98	1.718
15	311.86	545.98	1.751
25	499.44	899.53	1.801

TABLE 7.- STOPWATCH TIMINGS FOR NAVIER-STOKES EQUATION

Number of iterations	t_{MIMD}, sec	t_{serial}, sec	Speedup
1	43.73	58.39	1.335
2	68.58	106.75	1.557
5	151.50	246.11	1.642
10	269.69	479.39	1.778
25	647.78	1175.21	1.814

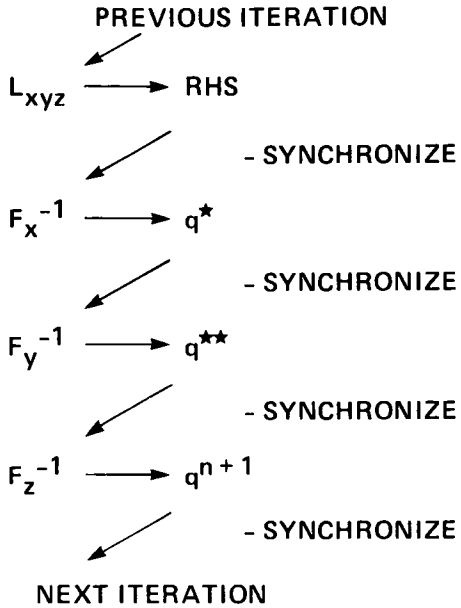

Figure 1.- MIMD procedure for solving approximate-factored algorithms.

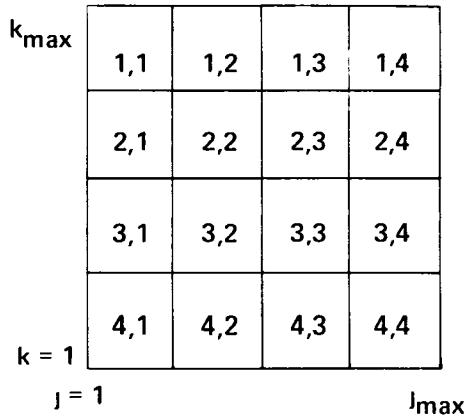

Figure 3.- Memory partition for a four-processor system set up for a two-dimensional problem.

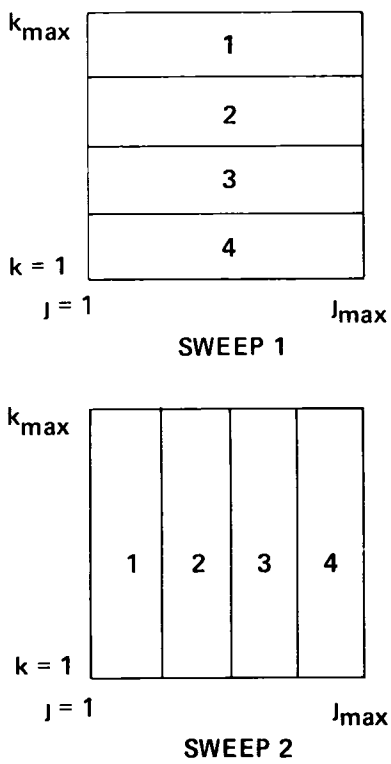

Figure 2.- Implementation of a two-dimensional problem on a four-processor system.

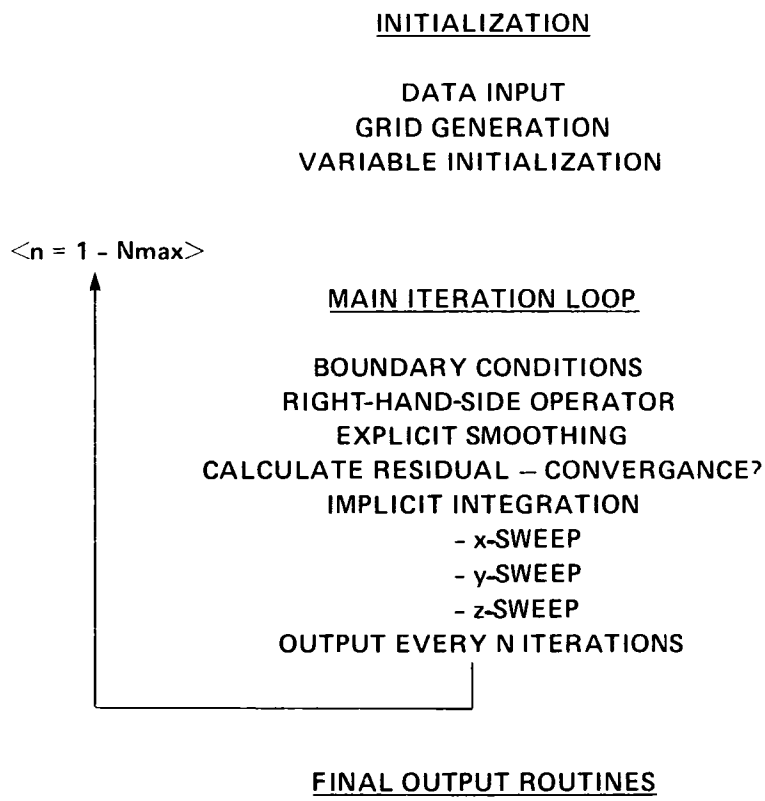

Figure 4.- Serial code flow summary.

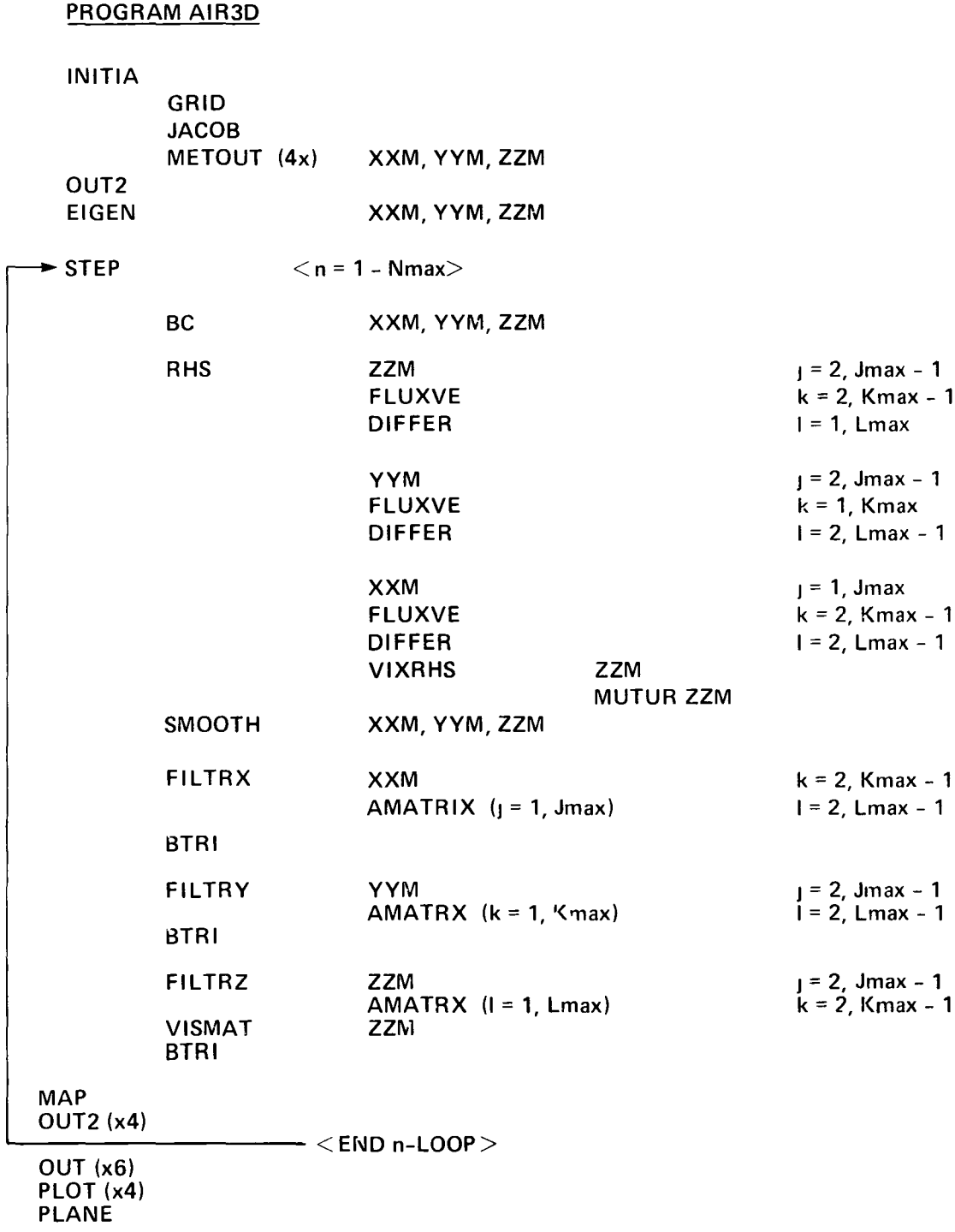

Figure 5.- Serial code subroutines.

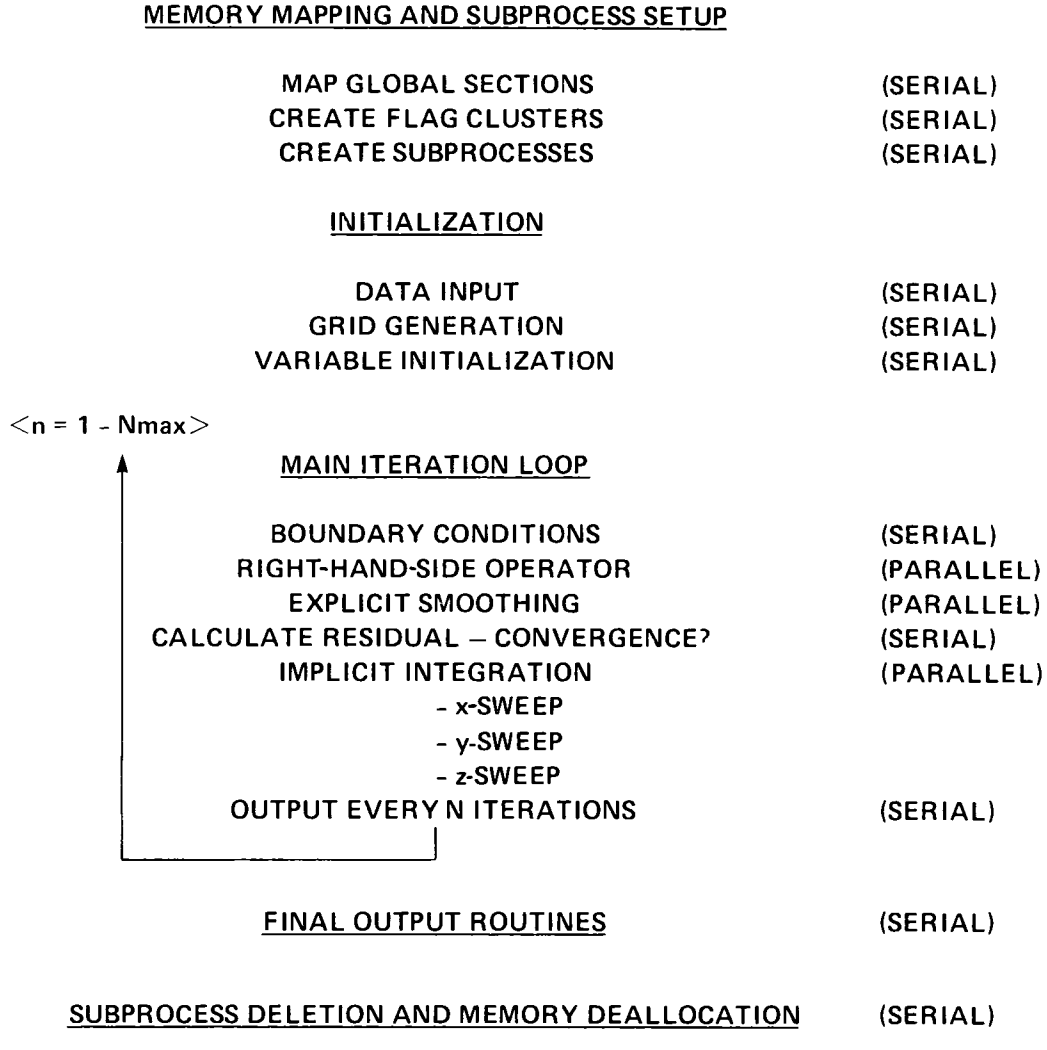

Figure 6.- Concurrent code flow summary.

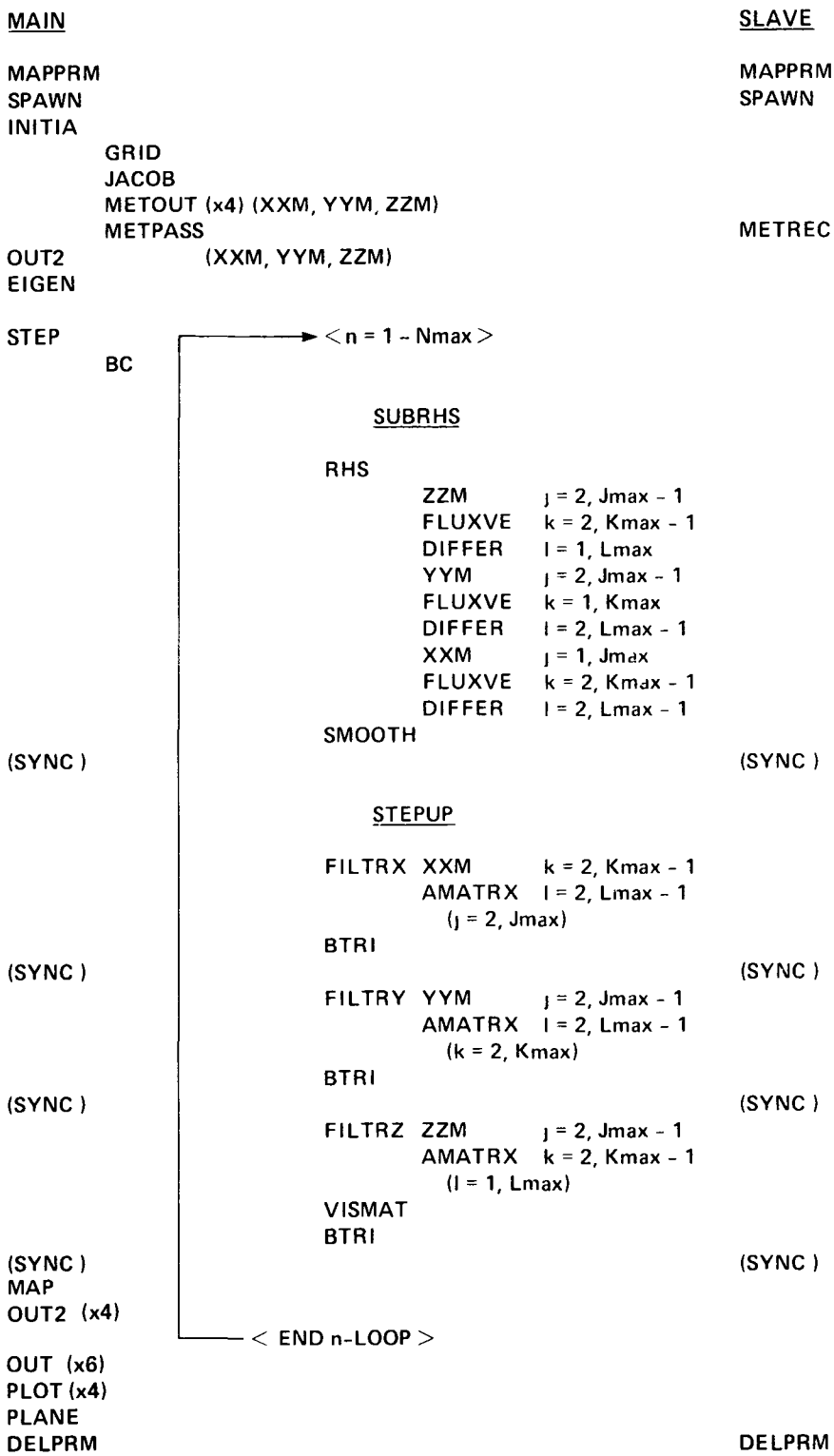

Figure 7.- Concurrent code subroutines.

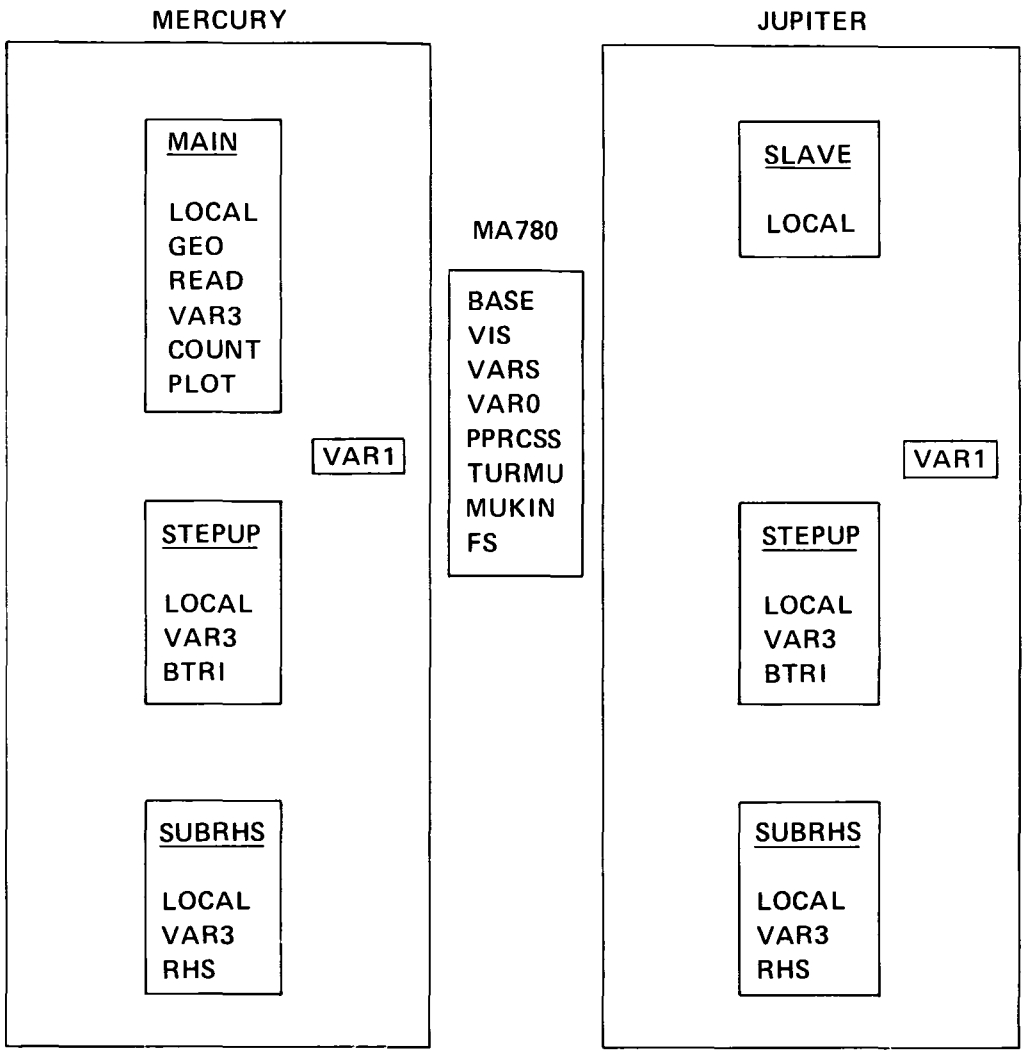

Figure 8.- Process/memory allocation (using data block names).

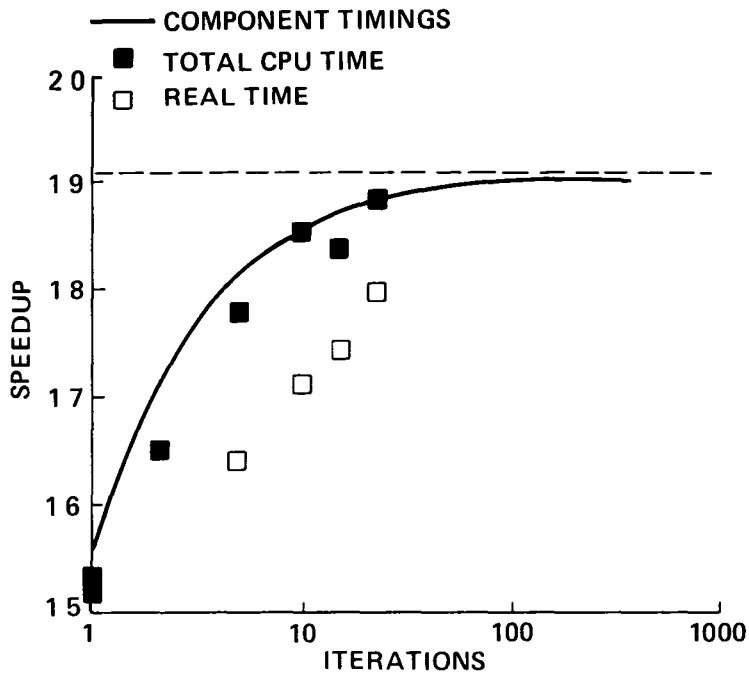

Figure 9.- Speedup of the Pulliam-Steger AIR3D code using two processors to solve the Euler equations.

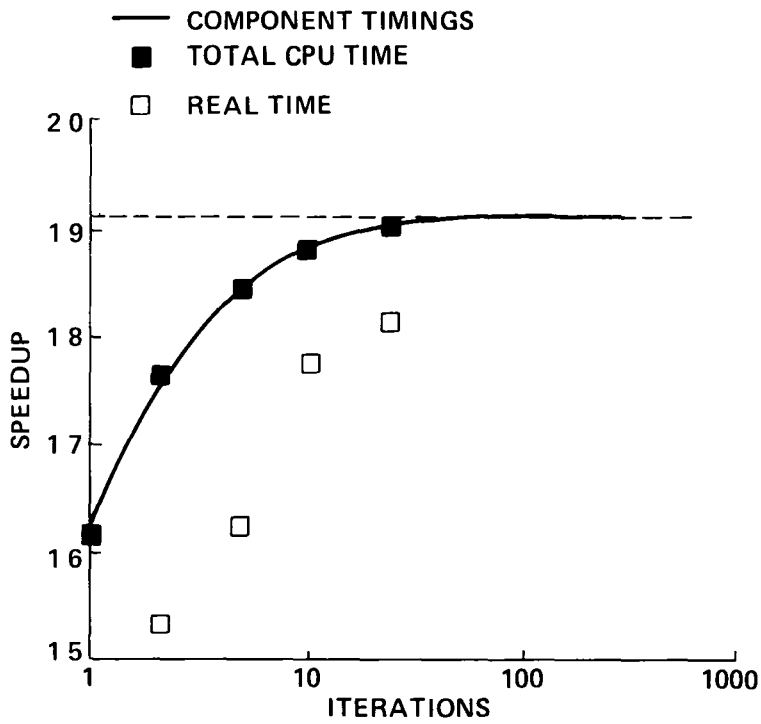

Figure 10.- Speedup of the Pulliam-Steger AIR3D code using two processors to solve the Navier-Stokes equation with a thin-layer approximation and an algebraic turbulence model.

1 Report No	2 Government Accession No	3 Recipient's Catalog No
NASA TM-85945		

4 Title and Subtitle	5 Report Date
STUDY OF THE MAPPING OF NAVIER-STOKES ALGORITHMS ONTO MULTIPLE-INSTRUCTION/MULTIPLE-DATA-STREAM COMPUTERS	6 Performing Organization Code July 1984
7 Author(s) D. Scott Eberhardt and Donald Baganoff (Stanford Univ., Stanford, CA), and Ken Stevens	8 Performing Organization Report No A-9716
9 Performing Organization Name and Address Ames Research Center Moffett Field, CA 94035	10 Work Unit No 11 Contract or Grant No
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546	13 Type of Report and Period Covered Technical Memorandum 14 Sponsoring Agency Code 505-37-01

15 Supplementary Notes

Point of Contact: Ken Stevens, Ames Research Center, MS 233-14, Moffett Field, CA 94035, (415) 965-5949 or FTS 448-5949

16 Abstract

Implicit approximate-factored algorithms have certain properties that are suitable for parallel processing. This study demonstrates how a particular computational fluid dynamics (CFD) code, using this algorithm, can be mapped onto a multiple-instruction/multiple-data-stream (MIMD) computer architecture. An explanation of this mapping procedure is presented, as well as some of the difficulties encountered when trying to run the code concurrently. Timing results are given for runs on the Ames Research Center's MIMD test facility which consists of two VAX 11/780's with a common MA780 multi-ported memory. Speedups exceeding 1.9 for characteristic CFD runs were indicated by the timing results.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
Navier-Stokes equations MIMD Concurrent processing Computational fluid dynamics	Unlimited Subject category - 62

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price*
Unclassified	Unclassified	26	A03

*For sale by the National Technical Information Service Springfield, Virginia 22161

End of Document